[MAC0211] Laboratório de Programação I
Aula 4
Linguagem de Montagem

Kelly, adaptado por Gubi

DCC–IME–USP

6 de agosto de 2017

Instruções aritméticas – multiplicação: **MUL**

Formato

Não tem o mesmo formato que as operações aritméticas anteriores porque a multiplicação pode gerar um número que tem até o dobro de bits que os operandos.

mul é válida apenas para a multiplicação de números sem sinal.

- **mul** ***reg/mem***
  Se o operando tem 8 bits, por exemplo,

  mul %bh

  então o comando equivale a

  ax <– al × bh

  Ou seja, o operando é sempre multiplicado pelo valor em al e o resultado é armazenado em ax.

Instruções aritméticas – multiplicação: **mul**

Formato

- **mul** *reg/mem*

  Se o operando tem 16 bits, por exemplo,

  mul %bx

  então o comando equivale a

  %dx:%ax <– %ax × %bx

  Ou seja, o operando é sempre multiplicado pelo valor em ax e o resultado de 32 bits é armazenado em 2 registradores de 16 bits: os 16 primeiros bits em ax e os 16 últimos em dx.

Instruções aritméticas – multiplicação: **MUL**

Formato

- **MUL** ***reg/mem***

  Se o operando tem 32 bits, por exemplo,

  mul %ebx

  então o comando equivale a

  edx:eax <– eax × ebx

  Ou seja, o operando é sempre multiplicado pelo valor em eax e o resultado de 64 bits é armazenado em 2 registradores de 32 bits: os 32 primeiros bits em eax e os 16 últimos em edx.

Obs.: O mul não pode ser usado com um valor constante. Por exemplo, o comando a seguir é inválido: mov $7

Instruções aritméticas – divisão inteira: **DIV**

Formato

Funciona de forma inversa ao mul.
div é válida apenas para a divisão de números inteiros sem sinal.

- **div** *reg/mem*

  por exemplo,

  div %bh

  divide o valor em ax pelo valor em bh, armazenando o quociente em al e o resto em ah

Divisor	Dividendo	Resto	Quociente
32 bits	edx:eax	edx	eax
16 bits	dx:ax	dx	ax
8 bits	ax	ah	al

Instruções aritméticas – divisão: **DIV**

Situações que geram exceção:

- divisão por zero
- transbordamento (*overflow*) – ocorre quando o resto gerado na divisão não cabe no registrador. Exemplo:

```
mov   $1024,%ax
mov   $2,%bh
div    %bh
```

Quociente deveria ser armazenado em AL, mas 512 ocupa no mínimo 10 bits!

Instruções aritméticas – divisão e multiplicação envolvendo números com sinal: **IMUL** e **IDIV**

Funcionam de modo análogo aos comandos DIV e MUL, mas podem ser aplicados a números com sinal.

Formato

- **imul** *reg/mem*
- **idiv** *reg/mem*

Instruções lógicas: **AND**, **OR**, **NOT**

O resultado é armazenado no segundo operando.

Formato

- **and** *reg/mem/const*, *reg* ou **and** *reg/const*, *mem*
- **or** *reg/mem/const*, *reg* ou **or** *reg/const*, *mem*
- **xor** *reg/mem/const*, *reg* ou **xor** *reg/const*, *mem*
- **not** *reg/mem* ; **inverte os bits**

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

NOT	0	1
	1	0

Exemplos

```
and   %bx,%ax   |   or   $0x5F,%cx   |   not   %ax
```

"Truques" com números binários

A operações lógicas podem ser usadas para:

- "resetar"/limpar (= atribuir zero a) bits
- "setar" (= atribuir 1 a) bits
- inverter bits
- examinar bits

Para "setar" um bit

Exemplo: setar o 3º bit menos significativo do AH.

```
or   $0b00000100, %ah
```

Para "resetar" um bit

Exemplo: resetar o 3º bit menos significativo do AH.

```
and   $0b11111011,%ah
```

"Truques" com números binários

Para inverter bits específicos

Exemplo: Inverter o quarto bit mais significativo do AX.

```
xor   $0x1000,%ax
```

Para examinar bits específicos

Exemplo: determinar o valor do quarto bit mais significativo do AX.

```
and   $0x1000,%ax
```

Se o resultado da operação for zero, o bit desejado vale 0. Senão, o bit vale 1.

"Truques" com números binários

Para zerar um registrador

Exemplo: zerar o registrador ECX.

```
xor   %ecx,%ecx
```

Para verificar se um registrador é nulo

Exemplo: verificar se ECX é nulo.

```
or   %ecx,%ecx
```

Obs.: se o registrado for nulo, então a *flag* zero é setada.

Instrução para trocar sinal – **NEG**

Gera o Complemento 2 do operando e armazena-o no próprio operando (ou seja, troca o sinal do operando).

Formato

- **NEG** ***reg/mem***

Exemplo

```
neg   %eax   <-->   not   %eax
                    add   $1,%eax
```

Instruções para a transferência de controle
Salto incondicional – **JMP**

Transfere a execução para o endereço especificado pelo rótulo

Formato:

- **jmp** ***rot***

Exemplo de programa

```
inicio:   mov  $5, %eax
          add  %eax, %eax

          jmp  inicio
```

Instrução para comparação – CMP

Compara o valor do primeiro operando com o valor do segundo.

Formato:

- **CMP** ***reg/mem/const, reg***

Resultado da comparação é armazenado em uma *flag*.

Exemplos

```
cpm     $5, %ax
cpm     (%ebx), %cx
```

Instruções para saltos condicionais

Variações:

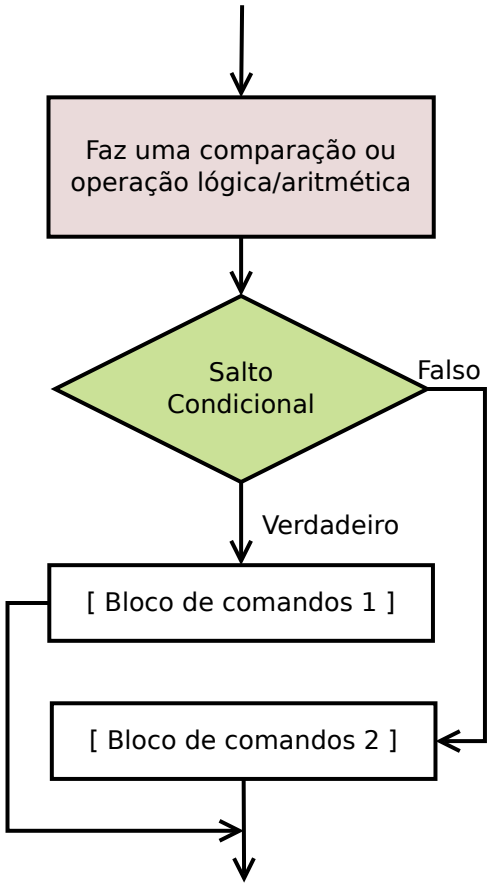
- **JE** – *jump if equal* (salta se é igual)
- **JNE** – *jump if equal* (salta se não é igual)
- **JG** – *jump if greater* (salta se é maior)
- **JGE** – *jump if greater or equal* (salta se é maior ou igual)
- **JNG** – *jump if not greater* (salta se não é maior)
- **JNGE** – *jump if not greater or equal* (salta se não é maior ou igual)
- **JL** – *jump if less* (salta se é menor)
- **JLE** – *jump if less or equal* (salta se é menor ou igual)
- **JNL** – *jump if not less* (salta se não é menor)
- **JNLE** – *jump if less or equal* (salta se não é menor ou igual)

Esses saltos consideram o resultado da última comparação realizada.

Importante: esses saltos consideram que a comparação envolveu números com sinal (*signed*).

Estrutura de um comando "if-else"

se (expressão) então «bloco 1» senão «bloco 2» fim;



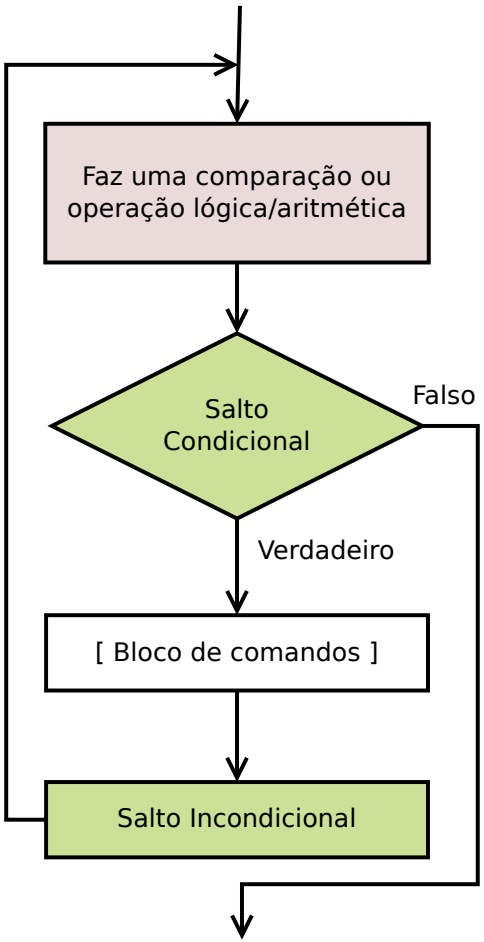
Exemplo de implementação em assembly

se (cont<15) então «bloco 1» senão «bloco 2»;

```
            :
            cmp   15,%CX
            jnl   senao       ; se contador >= 15
                              ; então vai para o bloco 2
            :                 ; bloco 1 de comandos
            jmp   fimse
            :
senao:      :                 ; bloco 2 de comandos
            :
fimse:      :                 ; instruções depois do SE
```

Estrutura de um comando "while"

enquanto (expressão) faça «bloco de comandos» fim;



Exemplo de implementação em assembly

while (cont<15) faça «bloco de comandos» fim;

```
            :
            mov     $0,%cx          ; inicializa o contador
inicio:     cmp     $15,%cx
            jge     fim             ; se contador >= 15,
                                    ; sai do laço
            :                       ; bloco de comandos
            inc     %cx             ; incrementa o contador
            jmp     inicio          ; vai para o início do laço
fim:        mov     (%ebx),%ax      ; 1ª instrução fora do laço
            :
```

Instruções para saltos condicionais – JZ e JNZ

Variações:

- **JZ** – *jump if zero* (salta se é nulo)
- **JNZ** – *jump if not zero* (salta se não é nulo)

Esses saltos consideram o resultado da última operação aritmética ou lógica realizada.

Instruções para saltos condicionais (versão *unsigned*)

Esses saltos consideram o resultado da última comparação realizada.
Consideram também que a comparação envolveu números sem sinal (*unsigned*).

Variações:

- **JA** – *jump if above* (salta se é maior)
- **JAE** – *jump if above or equal* (salta se é maior ou igual)
- **JNA** – *jump if not above* (salta se não é maior)
- **JNAE** – *jump if not above or equal* (salta se não é maior ou igual)
- **JB** – *jump if below* (salta se é menor)
- **JBE** – *jump if below or equal* (salta se é menor ou igual)
- **JNB** – *jump if not below* (salta se não é menor)
- **JNBE** – *jump if below or equal* (salta se não é menor ou igual)

Chamadas ao sistema operacional

Chamadas ao sistema (= *system calls*, ou somente *syscalls*)

- Forma por meio da qual programas solicitam serviços ao núcleo do SO
- Exemplos de serviços: operações para leitura e escrita em arquivos, criação e execução de novos processos, etc.

Chamadas ao sistema – como fazê-las em *assembly*?

- colocar número da chamada ao sistema em EAX
- colocar 3 primeiros argumentos em EBX, ECX, EDX (mais ESI e EDI se necessário)
- gerar a interrupção de chamada ao sistema (instrução INT 0x80)
- quando há valor de retorno, ele é colocado em EAX

Montadores

GCC Inline Assembly

- Suporte à arquitetura x86 bastante satisfatório
- Possibilita que código em linguagem de máquina seja inserido em programas em C
- Usa o GAS

GAS – *GNU Assembler*

- Por padrão, segue a sintaxe da AT&T (e não a da Intel, usada pela maioria dos montadores). Mas, em suas versões mais novas, aceita também a sintaxe da Intel
- Plataformas: Unix-like, Windows, DOS, OS/2
- Parte do pacote `binutils` do Linux
- Nome do executável: `gas` ou simplesmente `as`

Montadores

NASM – *Netwide Assembler*

- Bastante usado (confiável para o desenvolvimento de aplicações de grande porte, de uso comercial e industrial)
- Plataformas: Windows, Linux, Mac OS X, DOS, OS/2
- Instalação: pacote `nasm` do Linux

```
$ sudo apt-get install nasm
```

"Hello, world!" para NASM (versão 32 bits) – hello.asm

```
global _start        ; exporta para o ligador (ld) o ponto de entrada

section .text
_start:

    ; sys_write(stdout, mensagem, tamanho)

    mov eax, 4            ; chamada de sistema sys_write
    mov ebx, 1            ; stdout
    mov ecx, mensagem     ; endereço da mensagem
    mov edx, tamanho      ; tamanho da string de mensagem
    int 80h               ; chamada ao núcleo (kernel)

    ; sys_exit(return_code)

    mov eax, 1          ; chamada de sistema sys_exit
    mov ebx, 0          ; retorna 0 (sucesso)
    int 80h             ; chamada ao núcleo (kernel)

section .data
mensagem: db 'Hello, world!',0x0A     ; mensagem e quebra de linha
tamanho:  equ $ - mensagem            ; tamanho da mensagem
```

"Hello, world!" para GAS (versão 32 bits) – hello.S

```
.text

   .global  _start     # exporta para o ligador (ld) o ponto de entrada

_start:

    # sys_write(stdout, mensagem, tamanho)
    movl    $4, %eax              # chamada de sistema sys_write
    movl    $1, %ebx              # stdout
    movl    $mensagem, %ecx       # endereço da mensagem
    movl    $tamanho, %edx        # tamanho da string de mensagem
    int     $0x80                 # chamada ao núcleo (kernel)

    # sys_exit(codigo_retorno)
    movl    $1, %eax      # chamada de sistema sys_exit
    movl    $0, %ebx      # retorna 0 (sucesso)
    int     $0x80         # chamada ao núcleo (kernel)

.data
mensagem:
    .ascii  "Hello, world!\n"      # mensagem e quebra de linha
    tamanho =   . - mensagem       # tamanho da mensagem
```

Geração do executável

Passo 1 – Geração do código objeto

- Usando NASM, em um computador de 32 bits:

```
$ nasm -f elf32 hello.asm
```

- Usando NASM, em um computador de 64 bits:

```
$ nasm -f elf64 hello.asm
```

- Usando o GAS:

```
$ as -o hello.o hello.S
```

Os comandos acima gerarão um arquivo `hello.o`.

Geração do executável

Passo 2 – Ligação (geração do código de máquina)

```
$ ld -s -o hello hello.o
```

O comando acima gerará o arquivo executável `hello` .

Estrutura de um programa em linguagem de montagem

Seções

- `.text` – onde fica o código-fonte; é uma seção só para leitura
- `.data` – onde fica os dados/variáveis
- `.bss` – onde fica os dados/variáveis não inicializados

Bibliografia e materiais recomendados

- Capítulos 3, 4 e 6 do livro *Linux Assembly Language Programming*, de B. Neveln
- Livro *The Art of Assembly Language Programming*, de R. Hyde
  `http://cs.smith.edu/~thiebaut/ArtOfAssembly/artofasm.html`
- *The Netwide Assembler* – NASM
  `http://www.nasm.us/`
- *GNU Assembler* – GAS
  `http://sourceware.org/binutils/docs-2.23/as/index.html`
- *Linux assemblers: A comparison of GAS and NASM*
  `http://www.ibm.com/developerworks/linux/library/l-gas-nasm/index.html`
- Tabela de chamadas ao sistema no Linux
  `http://www.ime.usp.br/~kon/MAC211/syscalls.html`

Cenas dos próximos capítulos...

- Declaração de variáveis e constantes
- Mais exemplos de programas

Apêndice: "Hello, world!" para NASM (versão 64 bits)

```
global _start          ; exporta para o ligador (ld) o ponto de entrada

section .text
_start:

    ; sys_write(stdout, mensagem, tamanho)

    mov rax, 1             ; chamada de sistema sys_write
    mov rdi, 1             ; stdout
    mov rsi, mensagem      ; endereço da mensagem
    mov rdx, tamanho       ; tamanho da string de mensagem
    syscall                ; chamada ao núcleo (kernel)

    ; sys_exit(return_code)

    mov rax, 60            ; chamada de sistema sys_exit
    mov rdi, 0             ; retorna 0 (sucesso)
    syscall                ; chamada ao núcleo (kernel)

section .data
mensagem: db 'Hello, world!',0x0A     ; mensagem e quebra de linha
tamanho:  equ $ - mensagem            ; tamanho da mensagem
```